Towards a New Perspective on Partial Evaluation

(FUSE-MEMO-92-12 was published in the proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation, San Francisco, June, 1992)

Morry Katz
Computer Systems Laboratory
MJH 416A
Stanford University
Stanford, CA 94305
(katz@cs.stanford.edu)

Daniel Weise*
Computer Systems Laboratory
MJH 426
Stanford University
Stanford, CA 94305
(weise@cs.stanford.edu)

1 Introduction

We have designed a new method for performing partial evaluation. This method divides specialization into two phases: a polyvariant analysis phase and a code generation phase. The analysis phase does not differentiate between unfolding and specializing. *Symbolic execution* is the sole operation performed during the analysis phase. Symbolic execution of an expression yields a characterization of the value that would be returned by the expression if it were executed at runtime, a description of the residual operation(s) that must be performed to generate the runtime value, and a record of the information utilized in performing symbolic execution. (The third result of symbolic execution will be explained in some detail later.) Only delta-reductions are performed during symbolic execution. No beta-substitution of user functions occurs during the analysis phase and no values are in-lined. Consequently, symbolic execution yields extremely polymorphic, and highly reusable, *specialized function bodies*. The code generation phase constructs the residual program from the specialized function bodies. It is responsible for all beta-substitution. Delaying the beta-substitution decisions until the code generation phase allows us to construct a partial evaluator that both produces highly specialized code and makes intelligent decisions regarding code size versus lower function call overhead.

Termination decisions are made in the analysis phase based on *lazy use-analysis*, an extension of eager use-analysis [5]. Lazy use-analysis considers both the information used by symbolic execution in performing delta-reductions and how information about the values returned by delta reductions is used. Information is only used in a fundamental sense if there is a causal chain between its use in performing some delta reduction and the production of the final runtime answer returned by a program. The less information used in creating a specialized function body that contributes to the return value of the function, the greater the number of contexts in which the specialization can be reused. Furthermore, the less information used about the value returned by a function call, the less restrictions that are placed on the characteristics of the specialization to be called. A partial evaluator based on lazy use-analysis promises to produce a better combination of quality residual code and termination than previous alternatives. For example, a partial evaluator constructed using lazy use-analysis safely handles the standard "counter" problem[1] and properly unfolds Mogensen's (unpublished) regular expression accepter. No automatic partial evaluator to date has been able to do both.

The remainder of this paper is divided into three sections. The first presents a classification scheme for the types of divergences that partial evaluators experience and a discussion of different forms of termination (reuse) mechanisms. Next, use-analysis and its application to termination and reuse mechanisms are presented. The paper closes with our plans for future research and some conclusions.

2 Termination

2.1 Types of Divergence

We divide the types of divergence experienced by partial evaluators into three categories: *true divergences*, *hidden divergences*, and *induced divergences*. A program is considered divergent if normal execution of the program diverges for some valid program inputs. A partial evaluator is said to experience *true divergence* when it diverges processing a divergent input program and an input specification that includes values on which the input program diverges. Some convergent programs contain divergent segments that can never be reached for any valid set of inputs. If control ever reached these divergent pieces of code, the programs would diverge. We say such programs contain a *hidden divergence*. A partial evaluator may diverge because it stumbles across a hidden divergence during symbolic execution due to the partial nature of an input specification.

All other divergences of a partial evaluator are called *induced divergences*. The divergence does not exist in the input program, but is induced by the partial evaluation strategy being utilized. Induced divergences result from a partial evaluator pursuing a control path that could never be followed by a runtime evaluator. Some induced divergences result from bad termination strategies; others, from an inability of the partial evaluator to prove that a control path could never be taken.

*Supported in part by NSF Contract No. MIP-8902764, and in part by Defense Advanced Research Projects Agency Contract No. N0039-91-K-0138.

[1]Many partial evaluators fail to terminate on programs containing a recursion in which a static argument is incremented between each recursive call (e.g., counting up factorial) when the recursion cannot be completely unfolded during partial evaluation.

2.2 Types of Termination Mechanisms

We classify termination schemes based on induced divergences. *Termination priority (TP)* partial evaluators utilize termination strategies that ensure that no induced divergences will take place. *Residual code priority (RCP)* partial evaluators allow some induced divergences in an attempt to yield better residual code. In principle, a TP partial evaluator should be able to produce 'optimal' residual code in some cases. In practice, existing RCP partial evaluators produce better code than their TP cousins. We propose a partial evaluator based on a new form of analysis that is aggressive, yet attempts to minimize the classes of programs on which induced divergence will take place. Our reasons for taking the RCP approach are discussed in more detail later.

2.3 Essence of Termination

Termination is only an issue for recursive programs, as symbolic execution of non-recursive code always terminates. When a partial evaluator encounters a recursive function call,[2] the question to be answered is whether continued symbolic execution of the recursion will terminate. If symbolic execution will terminate, then it is safe for the partial evaluator to completely symbolically evaluate the recursion, and this will yield the best possible residual code. If symbolic execution will diverge, then it is desirable for a partial evaluator to discontinue symbolic execution and produce a residual loop. Unfortunately, it is not decidable whether symbolic execution will terminate.

All termination mechanisms use some concept of equivalence in deciding whether to continue symbolic execution or produce residual code. When a recursive call is reached, the current iteration of the loop is compared with previous iterations. If two iterations are deemed equivalent, then symbolic execution of the loop is discontinued and a residual loop is produced. If all iterations are deemed distinct, then symbolic execution is continued. TP partial evaluators use equivalence metrics that place all iterations of a loop into different equivalence classes only when symbolic execution of the loop terminates or the loop is truly divergent. RCP partial evaluators use equivalence metrics that only place two iterations of a loop in the same equivalence class if no improvement in residual code quality would result from continuing symbolic execution.

It is enlightening to think about termination of symbolic execution of recursions in terms of fixed points. Deciding whether to continue symbolic execution of a recursion is basically the same as determining whether symbolic execution of the recursion has reached a fixed point with respect to the creation of new, unique specialized function bodies. Once a fixed point has been reached, continued symbolic execution of the recursion would lead to divergence since the data available for making the termination decision is by definition not changing and the algorithm has previously decided based on this data to continue symbolic execution. The central issue, therefore, is how to define an equivalence metric that captures the concept of symbolic execution having reached, or not reached, a fixed point.

Two specializations are completely equivalent if one specialization can be replaced by the other (in any context) without changing a program's behavior. Complete equivalence is overly restrictive. A more liberal equivalence that gets to the essence of partial evaluation is: two specializations are equivalent with respect to a given call site in a residual program if a call to one specialization can be replaced by a call to the other without changing a program's behavior. Note that an original function and any nontrivial specialization of it are not equivalent in the first sense, but can be equivalent in the second.

[2]Not all partial evaluators distinguish recursive calls from non-recursive calls. This observation doesn't affect our argument.

2.4 Enlarging Equivalence Classes

Virtually all partial evaluators characterize specializations based upon the information in the arguments that the specializer was *allowed to use* when constructing the specialization.[3] For example, in offline specializers such as Similix [1] and MIX [4], specializations are characterized by the information in the static arguments; in online specializers such as Fuse [8], all of the information in the arguments characterizes the specialization. These characterizations form equivalence classes of arguments. A specialization can be safely reused at any call site where the information allowed to be used during specialization is present at the call site (i.e., wherever the arguments are in the equivalence class of the specialization). Regardless of how they are created or the context from which they are invoked, two specializations of the same function are equivalent when the equivalence classes of their arguments are equal.

Recently, Ruf showed how to enlarge the equivalence class (reusability) of a specialization by characterizing it by the information in the arguments that was *used to construct* the specialization [5]. He invented the term *domain of specialization* (DOS) to name the equivalence class a specialization can be safely reused on, and showed how to compute a safe approximation to the DOS (called the MGI) by tracking the information used to construct a specialization. Ruf showed that the larger equivalence classes that his methods constructed for a specialization significantly increased the reuse and sharing in residual code without degrading the quality of residual code.

The DOS defined by Ruf is not as large as possible because it assumes all the information contained in the value returned by a specialized function at runtime will be used; however, this is often not the case. The less information that is used about the return value of a specialization by some context, the larger the equivalence class of specializations that can be called from that call site. We will refer to the DOS defined in [5] as the *context free domain of specialization* (CF-DOS) to differentiate it from the *context sensitive domain of specialization* (CS-DOS) that accounts for both the values to which the specialization is applicable and how its results can be used. When a function is called from a context that doesn't use all the information available about its return value, it is important to know if a specialization already exists (even one returning a different value) that makes it reusable from this call site. The CF-DOS is not as useful for this purpose since it is a characterization based on information that may not be used in this context. That is, the CF-DOS may be too small to be of use, in particular, too small to ensure termination.

A direct consequence of the definition of the CS-DOS is that only information that *causally contributes to generating the result of a program* appears in the definition of any equivalence class. This follows from each specialization be-

[3]The significant exception to this rule is systems using manual annotations such as Schism [2].

ing characterized by the context in which it is called, which is in turn characterized by the context in which it is called, etc. We argue that the CS-DOS is the largest possible equivalence class that can be used by a partial evaluator without degrading the quality of residual code. (i.e., The CS-DOS characterizes specializations with the greatest potential for reuse that does not impact the ability to create useful specializations.)

To effectively approximate the equivalence classes of the CS-DOS, a partial evaluator should place two specializations of a function that depend on identical information from two separate argument vectors in the same equivalence class. Two otherwise equivalent specializations lose their equivalence if beta substitutions of values into the specializations yields residual code with differing equivalence classes. This means specialization must not perform unnecessary beta-substitution.

A partial evaluator that bases its control decisions on an accurate estimate of information usage can produce a better combination of residual code quality and termination properties than one that does not. For example, a TP partial evaluator could use knowledge of the information that would be used in forming a specialization to prove that symbolic execution of a function call would result in a new, unique specialized function body. This would enable a TP partial evaluator to be more polyvariant (i.e., create more useful specializations and perform more unfolding) while still maintaining its termination properties.

Conversely, an RCP partial evaluator that exploits the property that two specializations only fail to be interchangeable when different information about the available values was utilized in creating them can better decide whether symbolic execution of a function body will result in a new specialization. For example, whereas Similix assumes that progress is being made towards producing a new specialization as long as no introduced function is called with identical static arguments [1], an information based partial evaluator would only distinguish between argument sets that differed in some piece of information to be used in forming the specialization. This enables the RCP partial evaluator to correctly terminate more often.

2.5 Equivalence Classes and Termination

We propose using equivalence classes based upon use information as our sole termination method, unlike previous methods that appeal to *dynamic conditionals*, such as Fuse and Similix, use induction methods, such as Mix [6], use finiteness criteria, such as [3], or use manual annotations, such as Schism [2]. It is vital that we construct the largest classes possible without degrading residual code quality. In particular, to ensure termination, it is important that the equivalence class of a specialization not be restricted simply because the value of some variable has been inlined in the specialization, if inlining has not allowed for further optimization. Gratuitous inlining decreases the size of equivalences classes, and induces unnecessary distinctions between specializations.

For example, consider a procedure, called `check-numeric-type`, that signals an error if its sole argument is not numeric, and is otherwise the identity function (Figure 1). The application of `check-numeric-type` to the value `1` can yield two different specializations. Performing all possible beta reductions yields a specialization that just returns the value `1` (Figure 2). A more polymorphic specialization is the identity procedure that remains parameterized over the argument value, which must be numeric (Figure 3). A partial evaluator that generates the less polymorphic form of specializations could not reuse the specialization generated for `(check-numeric-type 1)` for a call to `(check-numeric-type 2)` in any context that uses the return value of the call.

```
(define check-numeric-type
  (lambda (val)
    (if (number? val)
        val
        (error val))))
```

Figure 1: A function for checking that an argument is numeric

```
(define check-numeric-type
  (lambda (val) 1))
```

Figure 2: A maximally specialized version of `(check-numeric-type 1)`

To detect the equivalences required for effective termination, we propose dividing the specialization process into two phases. During the first (analysis and symbolic execution) phase of partial evaluation, the specializer only incorporates into specializations that information about values that is necessary to perform *further optimizations* during symbolic execution. In other words, it would produce the more polymorphic specialization in Figure 3 for `(check-numeric-type 1)`. Only type information about the value `1` is used in evaluating the predicate of the conditional in `check-numeric-type`, and no information is used in the consequent. The specializer would not inline the value since doing so enables no further optimizations, and would restrict the applicability of the specialization (i.e., reduce the size of its equivalence class). Additional inlining decisions are delayed until the code generation phase. During code generation all of the call sites of a specialization are known, which determines the maximum amount of code sharing that is possible after the analysis phase has completed. At that time the traditional inlining trade-off between code size and execution efficiency can most effectively be made.

An added benefit of our two phase approach based on use-analysis is that the partial evaluator can choose to produce code for more or fewer specializations than it generated during the analysis phase. When a single specialization is only applied to a small number of elements of the equivalence class to which it is applicable, it may choose to generate a separate specialization for each element of the equivalence class that appears in the code. For example, a specialization might be applicable to all numbers, but only called with the arguments 0 and 1. Instead of producing a single specialization, it might be better to produce two specializations by inlining one of the possible argument values in each. This would yield two specializations: one applicable to 0 and the other applicable to 1. Conversely, the partial evaluator may occasionally combine several specializations

```
(define check-numeric-type
  (lambda (val) val))
```

Figure 3: A maximally polymorphic specialization of `(check-numeric-type 1)`

produced during the analysis phase into a single, more general specialization. An important property of use-analysis is that it produces code that is as polyvariant as desirable without forcing the code generation phase to produce more code than is necessary.

3 Use-Analysis

3.1 The Use-Analysis Framework

Use-analysis maintains an approximation to the information used by a partial evaluator during symbolic execution. Use information can be represented by points in an information lattice. Figure 4 shows the value domains for a pure subset of Scheme. A richer set of value domains is required for representing use information. They are needed because information about a value other than its identity is often used in forming a specialization. We add values to the Scheme domains that represent unspecified members of existing domains, similar to the *symbolic values* used by Fuse[8]. For example, use of only the integer property of the number 3 in a specialization might be represented by the abstract value $\bot_{Int}$. A complete set of value domains for a partial evaluator based on this concept is presented in Figure 5.

An information lattice based on the domains in Figure 5 and the binary relation $\prec$, meaning *has less information than*, is presented in Figure 6.[4] $\bot_{PEval}$ represents having no information about a value. The first clause in the lattice description states that there is information contained in the type of a value. The next three clauses express that there is more information in the identity of a value than in its type. Pairs are organized in an information hierarchy based on the information known about the car and cdr of the pair. The more information known about the two components of the pair, the more information that is known about the pair, itself. Finally, the last clause states that $\top$ represents more information than any other lattice element.

There are many types of information about values that cannot be precisely represented by nodes in the information lattice of Figure 6. For example, the best representation of the information that an integer is less than 5 is either the integer's identity or $\bot_{Int}$. The integer's identity is an overspecification, excluding other integers that are less than 5. $\bot_{Int}$ is an underspecification, including integers greater than or equal to 5. The effects on partial evaluation algorithms of different choices of information lattices and means of imprecisely representing information usage utilizing them will be discussed later.

Our presentation of use-analysis will proceed in three stages. First, an eager use-analysis that is fairly simple to explain and understand will be developed. A fully lazy use-analysis will then be discussed. Finally, lazy use-analysis that eliminates some of the shortcomings of eager use-analysis will be presented.

[4] The orientation of an information lattice is arbitrary in that one can either have more information represented by higher or lower points in the lattice. The orientation we have selected is opposite of the one used by Ruf in [5]

Eager Use-Analysis

A partial evaluator uses information about data values when performing computations during symbolic execution. Eager use-analysis, first presented in [5], records usage information as soon as information about a value is used in performing a computation. The information about a value that is utilized is represented by an element of an information lattice. Values in a partial evaluator are replaced by annotated values that are pairs composed of a representation of the information known about an object's value and a use annotation for that value. When the partial evaluator symbolically executes a primitive function on a set of annotated values, a new value is produced and the use information of the annotated argument values is updated. Use-analysis seeks to record the maximum amount of information about a value that is utilized, so primitives only modify the use information field of an annotated value when they use more information about that value than has previously been used.

A table of Scheme function applications and use profiles for their arguments based on the information lattice presented earlier (Figure 5) is shown in Figure 7. Less trivial use profiles result when multiple functions are composed. For example, `(number? (car (car '((1 2) (3 4)))))` results in a use profile that might be represented as $<< \bot_{Int}, \bot_{PEval} >, \bot_{PEval} >$. The innermost `car` uses the information that the argument is a pair. The next `car` uses the information that the first component of the pair is also a pair. `number?` uses the information that the first component of that pair is an integer.

Eager use-analysis often produces an overestimation of the information required to produce a given specialization. For example, in the expression `(number? (+ 1 2))`, `+` uses the identity of both of its arguments to produce the result 3 and modifies the use annotations of the arguments to reflect this usage. The function `number?` only uses that 3 is an integer, not its value. This function would return the same result for any two integer arguments to which `+` were applied. The specialization only depends upon the types of 1 and 2, not their identities (values). Embedding the example expression in a larger expression leads to yet greater overestimation of information usage. In the expression `((lambda (a b) a) #t (number? (+ 1 2)))`, no information about 1 or 2 is used since the result of `number?` is thrown away; however, the use annotations still show that the identities of these integers are used. While the code examples shown may seem unrealistic, similar expressions do appear in real programs because of macros and other abstraction mechanisms.

Fully Lazy Use-Analysis

Eager use-analysis has the flaw of recording information usage of intermediate computations that do not contribute to the result of a program. Ideally, only information utilized in generating the final result should be recorded. Information contributes to the result in one of two ways, either by affecting the data or the control flow of a program. How information about a value percolates through the data flow of a program to affect the final result is fairly obvious. Values also affect the result of a program when they are used to

Int	$=$	$0, \pm 1, \pm 2, \cdots$	integers
$Bool$	$=$	$\texttt{true} + \texttt{false}$	booleans
Nil	$=$	$\texttt{nil}$	empty list
$Pair$	$=$	$Sval \times Sval$	pairs
$Func$	$=$	$Sval^{\star} \rightarrow Sval$	function values
$Sval$	$=$	$Int + Bool + Nil + Pair + Func$	scheme values

Figure 4: Value domains for a subset of Scheme

Int	$=$	$0, \pm 1, \pm 2, \cdots$	integers
		$\bot_{Int}$	unspecified integer
$Bool$	$=$	$\texttt{true} + \texttt{false}$	booleans
		$\bot_{Bool}$	unspecified boolean
Nil	$=$	$\texttt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\bot_{Pair} = \bot_{PEval} \times \bot_{PEval}$	unspecified pair
$Func$	$=$	$PEval^{\star} \rightarrow PEval$	function values
		$\bot_{Func}$	unspecified function
$Kval$	$=$	$Int + Bool + Nil + Pair + Func$	known values
$Bots$	$=$	$\bot_{Int} + \bot_{Bool} + \bot_{Pair} + \bot_{Func}$	bottom values
$PEval$	$=$	$Kval + Bots + \bot_{PEval}$	partial evaluation values
		$\bot_{PEval}$	unspecified value

Figure 5: Value domains for information

$$\forall x \in (Bots \cup Nil).(\bot_{PEval} \prec x)$$
$$\forall i \in Int.(\bot_{Int} \prec i)$$
$$\forall b \in Bool.(\bot_{Bool} \prec b)$$
$$\forall f \in Func.(\bot_{Func} \prec f)$$
$$\forall <x,y> \in Pair.(\forall x' \prec x.(<x',y> \prec <x,y>))$$
$$\forall <x,y> \in Pair.(\forall y' \prec y.(<x,y'> \prec <x,y>))$$
$$\forall <x,y> \in Pair.(\forall x' \prec x.(\forall y' \prec y.(<x',y'> \prec <x,y>)))$$
$$\forall k \in (Kval \cup Bots).(k \prec \top)$$

Figure 6: Information lattice description

Expression	Argument Use Profiles
`(+ 1 2)`	1,2
`(number? 3)`	$\bot_{Int}$
`(car '(1 . 2))`	$\bot_{Pair}$
`(boolean? #t)`	$\bot_{Bool}$

Figure 7: Eager use annotations

make a control flow decision (e.g., the predicate of a conditional). If a control flow decision affects the final result of a program, then the information used to make the control flow decision has been used in generating the program's result.

In fully lazy use-analysis a complete model of the execution of the entire program must be built before a use-analysis can be performed. Only then can it be determined if there is a causal chain from the use of some information at one point in a program's execution all the way to the final result. While fully lazy use-analysis would be ideal, it is unfortunately unknown at this time how to implement it, or even if it is theoretically computable.

Lazy Use-Analysis

Lazy use-analysis is an approximation to fully lazy use-analysis that solves many of the problems associated with eager use-analysis. Lazy use-analysis delays deciding whether information about values has been used until the information usage contributes to making a control flow decision. It differs from its fully lazy cousin in that all control flow decisions immediately assert the usage of information, even if they eventually turn out not to contribute to the final result. Lazy use-analysis works backwards from a point where information is used to make a control flow decision back through the data flow to the original sources of the information used in making that decision. Lazy use-analysis maintains a record relating the information usage of one value to the information usage of another value. For example, execution of the expression `(number? 3)` creates a link between using the value of the boolean result and using the type of `3`. Execution of `(+ 1 2)` creates links that express two properties: use of the value of the result implies use of the values of both `1` and `2`, and use of the type of the result implies use of the types of both arguments (i.e.,, that they are integers).

When a value affects a control flow decision, lazy use-analysis records usage information about that value using an annotation from an information lattice. If the value utilized in the control flow decision is linked to other values, then the values to which it is linked also have information usage recorded about them. They, in turn, may pass on usage information to other values to which they are linked. This process is continued until all necessary values have had their usage profiles updated. Since the use links follow the data flow of the program, they must be acyclic and the process is guaranteed to terminate.

3.2 Termination Based on Use-Analysis

Our termination mechanism defines its equivalence classes in terms of information usage profiles. When the first recursive call to a function is detected, an information based partial evaluator has already acquired information about the first iteration of the loop. However, the partial evaluator must continue symbolic execution until a second recursive call is made, because two equivalent calls (iterations) are required to detect a fixed point. Once the second recursive call is reached, the question is whether the two iterations are equivalent with respect to information usage. If the iterations are equivalent, then the analysis process has reached a fixed point, and a residual loop will be produced. If the iterations are not equivalent, then symbolic execution should continue until either the recursion terminates (i.e., is completely unfolded) or a new iteration which is equivalent to some previous iteration is detected.

We divide each recursive procedure into two segments. The *head* is the portion of the procedure performed before the recursive call; the *tail*, the portion performed after. The termination algorithm described so far has only utilized use information about the head of a recursion in deciding whether two iterations are equivalent. In attempting to produce a specialized function body, it may be determined that two iterations are in actuality not equivalent because their tails are not equivalent. In this case, symbolic execution of the recursion is resumed and proceeds as outlined above. A final residual, recursive loop is only produced when both the head and the tail of two iterations of a recursion are found to be equivalent.

An ideal partial evaluator maintaining perfect use information would suffer from no induced divergences and could produce as good residual code as is possible from any physically realizable system. Perfect use information would require a fully lazy analysis based on a set of value domains that could capture precisely the information used by all primitives in the language being specialized. This type of optimality arises because fully lazy use-analysis captures perfectly the applicability of all specializations based on the available data. Therefore, when symbolic execution of a recursion reaches a fixed point (finds two equivalent iterations) based on this perfect form of use-analysis, it is guaranteed that two iterations are fundamentally equivalent and that no alternative means of specialization could cause them to be nonequivalent.

Imperfect use profiles result in either TP or RCP partial evaluators. Information profiles that always over-record information usage yield RCP partial evaluators. Over-recording information usage can cause two equivalent iterations of a loop to appear distinct. This means the partial evaluator may not recognize that the symbolic evaluation process has reached a fixed point, and may diverge trying to completely unroll the recursion. Under-recording information usage yields a TP partial evaluator. Iterations of a recursion that are not in actuality equivalent may appear to be equivalent to such a partial evaluator. This means that an under-recording partial evaluator may give up unrolling a loop before the symbolic execution process has actually reached a fixed point, which results is an overly general equivalence class and a suboptimal specialization. However, under-recording can never lead to more symbolic execution than would result with perfect information profiles, so it must terminate whenever the ideal information based partial evaluator does. Since an ideal partial evaluator does not suffer from induced divergence, an under-recording partial evaluator will not, either.

4 Two Termination Examples

Lazy use-analysis enables a partial evaluator to produce higher quality code than many existing partial evaluators while incurring fewer induced divergences than other RCP partial evaluators. This is demonstrated through two example programs. No other automatic partial evaluator both terminates on the Iota program and produces good residual code for the Regular Expression Acceptor.

```
(define iota
  (lambda (n)
    (define loop
      (lambda (i)
        (if (= i n)
            '()
            (cons
             i
             (loop (1+ i))))))
    (loop 0)))
```

Figure 8: Iota function

4.1 Effective Termination

Partial evaluation of the `iota` function in Figure 8 with respect to an unknown value for `n` causes an induced divergence for Similix[1]. This system defines equivalence in terms of the identity of the values to which parameters that have been labeled as *static* by a *binding time analysis*[4] are bound. The salient issue is that it parameterizes its specialization points by the variable `i`, which it labels as being static. Since the value of `i` is incremented on every recursive call to `loop`, no two iterations of the recursion ever have identical values for `i` so Similix unfolds and symbolically executes `loop` forever.

Partial evaluation of `iota` based on lazy use-analysis terminates. During the first symbolic execution of `loop`, no use annotation is created for `(= i n)` since `n` is unknown; and, no use annotation is created for evaluating `i` to place it in the car of a cons cell since this involves no computation. The expression `(1+ i)` causes the creation of a use link between the value returned by the function application and the value of `i`. The second iteration creates a similar annotation, but no actual use of `i`. At the point when symbolic execution reaches the second recursive call to `loop`, it is recognized that the two previous iterations of the loop have used identical information about the free variables (i.e., no information). In other words, the two iterations of `loop` are equivalent so a residual loop is produced. Eventually, partial evaluation terminates yielding a residual program that is identical to the input program.

If `iota` had been applied to a known value, `(= i n)` would create a use dependence between the value of `i` and the resultant value of applying `=` to its arguments. The value of the predicate of the conditional would be used in making a control flow decision causing the value of the `=` expression, and therefore the value of `i`, to be used. Since each iteration of `loop` has a different value of `i`, no two iterations are ever found to be equivalent, and the recursion continues to be unfolded until the execution of `iota` is completed. The residual program that is eventually produced is just straight line code that conses together the list to be returned by `iota`.

4.2 High Quality Residual Code

Optimal residual code for partial evaluation of the regular expression matcher in Figure 9 with respect to a known regular expression, a^*, and an unknown input string is a completely inlined decision tree with a single loop for kleene star matching as shown in Figure 10.[5,6] Mix cannot produce this code because it only continues symbolic evaluation of a recursion as long as the loop is self recursive and all of the static arguments either have identical values or are bound to proper substructures of previous values. Leaving aside the self recursive limitation, the difficulty is that as a regular expression is processed, the expression is on average shrinking; however, kleene star processing temporarily increases the size of the regular expression. Also, the regular expression is often stored in two pieces, one of which is shrinking, and the other is growing.

Fuse uses a slightly more aggressive unfolding strategy. Unfolding and symbolic execution of a recursion is always continued unless two iterations of a recursion are separated by a *dynamic conditional,* one whose predicate is not decidable during partial evaluation. When a recursion spans a dynamic conditional, symbolic execution is still continued as long as all the arguments are either identical or shrinking in size. Although Fuse fails to produce optimal residual code for reasons similar to Mix, we include a more detailed outline of the steps performed by Fuse because it illuminates the underlying problem. Symbolic execution of the regular expression matcher by Fuse would proceed as follows:

```
(match? (make-kleene-star (make-term 'a)) ?)
(match? [kleene-star [term 'a]] ?)
(match-pattern? [kleene-star [term 'a]]
                [null-pattern] ?)
(match-star? [kleene-star [term 'a]]
             [null-pattern] ?)
(if (match? [null-pattern] ?)
    #t
    (match-pattern? . . .))
(match? [null-pattern] ?)
(match-pattern? [null-pattern] [null-pattern] ?)
(match-null? [null-pattern] ?)
(null? ?)
```

Since `(null? ?)` can not be evaluated during partial evaluation, this expression would be left residual. As a result, the conditional in `match-star?` would be dynamic. Symbolic execution of the alternative of this dynamic conditional would yield a call of the form `(match-pattern? [term 'a] [concat [kleene-star [term 'a]] [null-pattern]] ?)`. Since this is a recursive call to `match-pattern?` that spans a dynamic conditional and one argument grows in size, Fuse would suspend symbolic execution at this point and use generalization to produce a specialized version of the loop. The result is residual code that still includes the dispatcher in `match-pattern?` and many of the data-structure abstractions of the pattern matcher.

Partial evaluation of the regular expression matcher using lazy use-analysis is not inherently complicated, but it is easy to get lost in the details. This presentation will therefore proceed through a simulation of the partial evaluation

[5]The regular expression matcher presented does not include disjunctions or epsilon rules. These have been omitted for simplicity and do not effect the way in which any of the systems discussed would perform partial evaluation.

[6]Brackets are used in code segments to denote objects to which functions are to be applied. Question marks are used to represent unspecified values (i.e., $\perp_{PEval}$).

```
(define match?
  (lambda (pattern input)
    (match-pattern? pattern null-pattern input)))

(define match-pattern?
  (lambda (pattern rest-pattern input)
    (cond ((null-pattern? pattern)
           (match-null? rest-pattern input))
          ((term? pattern)
           (match-term? pattern rest-pattern input))
          ((kleene-star? pattern)
           (match-star? pattern rest-pattern input))
          ((concat? pattern)
           (match-concat? pattern rest-pattern input)))))

(define null-pattern?
  (lambda (pattern) (eq? pattern null-pattern)))

(define match-null?
  (lambda (rest-pattern input)
    (if (null-pattern? rest-pattern)
        (null? input)
        (match? rest-pattern input))))

(define match-term?
  (lambda (term-pattern rest-pattern input)
    (if (and (pair? input)
             (equal? (term-symbol term-pattern)
                     (car input)))
        (match? rest-pattern (cdr input))
        #f)))

(define match-star?
  (lambda (star-pattern rest-pattern input)
    (if (match? rest-pattern input)
        #t
        (match-pattern?
         (kleene-star-expr star-pattern)
         (concat star-pattern rest-pattern)
         input))))

(define concat
  (lambda (pattern1 pattern2)
    (cond ((null-pattern? pattern1) pattern2)
          ((null-pattern? pattern2) pattern1)
          (#t (make-concat pattern1 pattern2)))))

(define match-concat?
  (lambda (concat-pattern rest-pattern input)
    (match-pattern?
     (concat-head concat-pattern)
     (concat (concat-tail concat-pattern) rest-pattern)
     input)))
```

Figure 9: Regular expression matcher example

```
(define match?
  (lambda (pattern input)
    (if (null? input)
        #t
        (if (and (pair? input)
                 (equal? 'a (car input)))
            (match? [kleene-star [term 'a]]
                    (cdr input))
            #f))))
```

Figure 10: Optimal residual code for `(match? (make-kleene-star (make-term 'a)) ?)`

process virtually function call by function call. The procedure to which a recursive call will be detected and which will lead to termination is `match?`. This is the very first function that is called after the regular expression that is supplied as input has been generated.

`(match? (make-kleene-star (make-term 'a)) ?)` Initial call to begin partial evaluation.

`(match? [kleene-star [term 'a]] ?)` Regular expression description created.

`(match-pattern? [kleene-star [term 'a]] [null-pattern] ?)` A use dependence is created between the `pattern` variable in `match-pattern?` and the `pattern` variable in `match?`.

`(null-pattern? [kleene-star [term 'a]])` A use dependence is created between `pattern` in `null-pattern?` and `match-pattern?`.

`(eq? [kleene-star [term 'a]] [null-pattern]])` A use dependence is created between the `#f` returned by `eq?` and `pattern` in `null-pattern?`. The value of `#f` is used to make a control flow decision in the `cond` in `match-pattern?`. This use propagates along the use dependence links eventually asserting use of the identity of `pattern` in both `match-pattern?` and `match?`.

`(term? [kleene-star [term 'a]])` A use dependence is created between the `#f` returned by `term?` and `pattern` in `match-pattern?`. When this `#f` is used by the `cond` the same use assertions created by the previous clause of `cond` are made.

`(kleene-star? [kleene-star [term 'a]])` Same as the previous step.

`(match-star? [kleene-star [term 'a]] [null-pattern] ?)` A use dependence is created between the corresponding variables of `match-star?` and `match-pattern?`.

`(match? [null-pattern] ?)` This expression returns an unknown value. Since it is the predicate of a conditional, both arms of the conditional will be investigated by the partial evaluator. None of the use-analysis done for this expression is of interest, so the details have been omitted.

`(kleene-star-expr [kleene-star [term 'a]])` A use dependence is created between the value `[term 'a]` returned by the function application and the corresponding subcomponent of `star-pattern`.

`(concat [kleene-star [term 'a]] [null-pattern])` Use dependencies are created between the values `pattern1` and `pattern2` in `concat` and the values `star-pattern` and `rest-pattern` in `match-star?`. The values of both `pattern1` and `pattern2` are then used causing use assertions to be propagated back to `pattern` and `rest-pattern` in `match-pattern?`.

`(match-pattern? [term 'a] [kleene-star [term 'a]])` Use dependencies are created between the formal parameters in `match-pattern?` and the sources of the actuals. `null-pattern?` is then handled similarly to the previous call to this function.

`(term? [term 'a])` A use dependence is created between the `#t` that is returned and `pattern`. The `#t` is then used by the `cond` causing a use assertion to propagate back all the way to the `[term 'a]` subcomponent of the very first call to `match?`.

`(match-term? [term 'a] [kleene-star [term 'a]] ?)` Use dependencies are created between the corresponding arguments. The predicate in `match-term?` does not yield a value during partial evaluation so both arms of the conditional must be investigated. The use assertions made during the evaluation of the predicate are again uninteresting.

`(match? [kleene-star [term 'a]] ?)` This is the first recursive call to `match?`. At this point a use profile for the first iteration of the recursion is available. It shows that the value of `pattern` was used, but no information about `input` was required.

The second call to `match?` would proceed exactly like the first, yielding a second recursive call to this function. At that point, the use profiles of the two iterations would be compared. In both iterations, the value of `pattern` was used; however, since the values were identical, the two iterations are equivalent. As a result, a residual loop would be created for match. The residual code produced is precisely the optimal code for `(match? [kleene-star [term 'a]] ?)` shown in figure 10.

5 Future Research

We are in the process of building an RCP partial evaluator based on lazy use-analysis and an information lattice similar to that in Figure 6. The input language that the partial evaluator will accept is a subset of higher-order, pure (functional) Scheme including only the integer, boolean, pair, and closure data types. This language includes all of the fundamental complexities inherent in any higher-order, applicative order, functional language.

Our goals are to investigate the classes of programs on which the proposed mechanism terminates, how efficiently the mechanism can be implemented; and, depending on these results, in what ways the information lattice or information retention mechanisms might be modified to produce a better partial evaluator. We also hope to gain insight into the use of use information as a guide to partial evaluation. This may enable us to investigate TP use information based partial evaluators in the future.

6 Conclusions

A partial evaluator based on lazy use-analysis will terminate on programs including counters and will properly unfold Mogensen's (unpublished) regular expression accepter. No automatic partial evaluator to date has been able to do both. Use-analysis offers a potential solution to other open challenges in partial evaluation such as selecting which specializations to include in a residual program, specializing imperative programs, and performing driving. Selecting which specializations to produce in a residual program was the first application to which use-analysis was applied [5]. This work used an eager form of use-analysis and demonstrated the viability and effectiveness of use-analysis. We believe that lazy use-analysis will only improve the solution to this problem.

Partial evaluation of imperative programming languages is not conceptually complicated. The problem is how to determine when two iterations of a recursion are equivalent. Previous partial evaluators could not handle arbitrary side effects because the only known method for comparing two iterations was comparison of the complete stores at two points in the execution sequence. This is simply infeasible. In a use-analysis based partial evaluator, only those portions of the store that are used must be compared to decide equivalence. Since the portion of a store used by a program segment will typically be a small fraction of the entire store, use-analysis makes partial evaluation of imperative languages both computationally and conceptually feasible.

Whether use information can be utilized to perform driving [7] is an open question. Preliminary investigation indicates that maintaining use information not only about values, but also about the residual code produced, may enable a use based partial evaluator to perform the same optimizations achieved through driving. The viability of achieving driving through use-analysis should become clearer through further investigation of use information.

In conclusion, use-analysis is a promising new technology for solving many of the open problems in partial evaluation. We are in the process of building an RCP partial evaluator based on lazy use-analysis that will produce a better combination of residual code quality and effective termination than any existing system. We believe this system will demonstrate the efficacy of basing termination on use-analysis and act as a stepping stone to the solution of several other problems currently impeding the transition of partial evaluation from a laboratory experiment into a widely used tool.

References

[1] Anders Bondorf and Olivier Danvy. Automatic autoprojection for recursive equations with global variables and abstract data types. DIKU Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990.

[2] Charles Consel. New insights into partial evaluation: the SCHISM experiment. In *Proceedings of the 2nd European Symposium on Programming*, pages 236–246. Springer-Verlag, LNCS 300, 1988.

[3] Carsten Kehler Holst. Finiteness analysis. In *Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 473–495. Springer-Verlag Lecture Notes in Computer Science, August 1991.

[4] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 1(3/4):9–50, 1988.

[5] Erik Ruf and Daniel Weise. Using types to avoid redundant specialization. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, CN, June 1991.

[6] Peter Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.

[7] Valentin Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[8] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online program specialization. In *Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 165–191. Springer-Verlag Lecture Notes in Computer Science, August 1991.